<알고리즘 요약 정리본>
5장_2 레드블랙트리(Red-Black Tree)

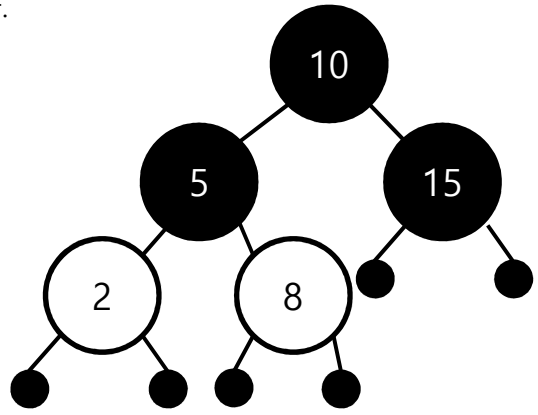
이번에는 레드블랙트리 부분을 파워포인트로 작성한 점에 대해서 양해를 구한다. 왜냐면 레드블랙트리에 대해서는 아시다시피 그림을 그리는 부분이 워낙 많기 때문에 이렇게 정리를 하지 않은 이상 비효율적이기 때문에다. 그래서 이번에는 그림을 통해서 레드블랙트리가 어떤 원리로 돌아가는지에 대해서 알아보는 기회를 가져보도록 하겠다.

0. 자가 균형 이진 탐색 트리

레드블랙트리에 들어가기 앞서서 자가 균형 이진 탐색 트리에 대해서 알고 넘어가면 좋겠다. 우리가 이진 탐색 트리에 대해서 임의의 값들을 계속 삽입하는 경우에는 보통의 경우에는 완전 이진 트리로 만들어지면서 완성이 되는 원리이지만, 만의 하나에 경우에는 편향 이진 트리가 나오게 되어 시간 복잡도가 O(n)이 되어버리는 최악의 경우도 고려를 해야 된다. 그래서 편향 이진 트리처럼 최악의 경우를 미리 방지하기 위해서 이진 탐색 트리의 시간 복잡도를 O(log n)으로 유지하도록 하는 트리를 자가 균형 이진 탐색 트리라고 한다. 여기서 레드 블랙트리도 자가 균형 이진 탐색 트리에 속한다고 볼 수 있다.

1. 레드블랙트리의 정의

우리가 색칠공부를 어렸을 때 많이 해봤을 것이다. 한 면에 검은색을 칠하면, 다른 면에 대해서 구분하기 쉽도록 다른 색을 칠해가는 원리로 채색을 한 경우에 대해서 생각을 해봤을 것이다. 이처럼 트리에도 컬러를 입혀서 값들이 추가되는데 있어서 임의로 추가되는 것이 아닌 색상의 규칙에 따라서 삽입이 되는 원리를 바로 레드블랙트리라고 볼 수 있겠다. 이 개념은 무려 1970년대 남진이 저 푸른 초원 위에를 부르던 시절에 나왔으니 40년 남짓한 개념으로 볼 수 있겠다. 레드 블랙트리의 사례는 아래처럼 볼 수 있겠다.



혹여 컬러 복사가 안 되는 사람들을 위해서 검은색 노드에 대해서는 음영 처리를 하였고, 빨간색 노드에 대해서는 하얀색으로 표기를 했으니, 하얀 부분에는 여러분들이 빨간색 팬을 준비해서 동그라미로 구분을 해두면 좋겠다. 레드 블랙트리에 대한 알고리즘을 접근하기 위해서는 아래와 같은 규칙에 대해서 알아두고 넘어갈 필요가 있다.

<레드블랙트리의 특성>

-> 레드블랙트리는 이진 탐색 트리의 알고리즘이 적용이 되는데 색상에 대한 정보(black, red)에 대해서 담고 있다.

-> 루트 노드는 언제나 검은색이다.
루트 노드는 아시다시피 위에 그림으로 보면 10이다.

-> 모든 리프 노드는 검은색이다.
여기서 리프 노드는 2, 8, 15라고 생각하면 큰 오산이다. 그 아래에 있는 좀만한 검은 동그라미가 바로 리프 노드이다. 이는 null 포인터가 NULL이라는 리프 노드를 가리킨다고 보면 되겠다.

-> 빨간색 노드의 자식은 반드시 검은색이 나온다.
레드블랙 트리에서 빨간색 노드의 의미는 막 삽입된 노드라고 보면 되는데 이들이 뭉쳐있으면 보기 불안정하게 보이기 때문에 이에 대해서 작업을 할 필요가 있다. 이는 레드블랙트리의 재조정 규칙을 살펴보도록 하자.

-> 루트 노드에서 임의의 리프 노드까지 거치는 경로에서 검은색 노드를 경유하는 수는 같다.
2, 8, 15의 자식들인 검은 동그라미(null 포인터)에서 5나 15를 즈려밟고 10까지 도달하는데 있어서 검은색 노드를 만나는 수는 모두 2이기 때문에 이에 대해 만족한다.

이처럼 레드블랙트리를 만족하기 위해서는 레드블랙트리의 재조정 규칙을 살펴볼 필요가 있다. 이에 대해 바로 설명하겠다.

2. 레드블랙트리의 삽입 과정과 재조정 규칙

레드블랙트리의 삽입 과정은 아까 레드블랙트리의 규칙처럼 그대로 적용을 하면 되는데 어떻게 적용하는지에 대해 공부를 하면서 알아보는 기회를 갖기 전에 레드와 블랙의 의미를 짚고 넘어가도록 하자.

2-0) 레드와 블랙의 의미

레드 노드 : 막 새로이 삽입된 노드. 혹은 블랙 노드였는데 재조정 규칙으로 인해 변하는 경우도 있다. 쉽게 이야기하면 직장에 들어온 신입사원과 같은 원리로 생각할 수 있겠다.

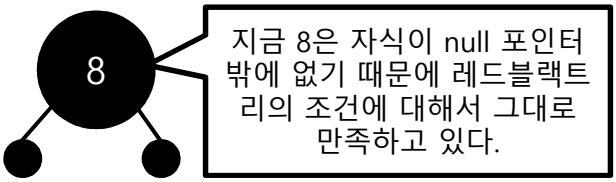
블랙 노드 : 자기 자리를 그대로 유지하려는 노드. 재조정 규칙으로 인해 레드 노드로 변할 수도 있다. 마치 회사에서 구조조정에 대한 두려움이 있는 대리 양반으로 생각하면 쉽겠다.

2-1) 레드 블랙 트리의 삽입 과정

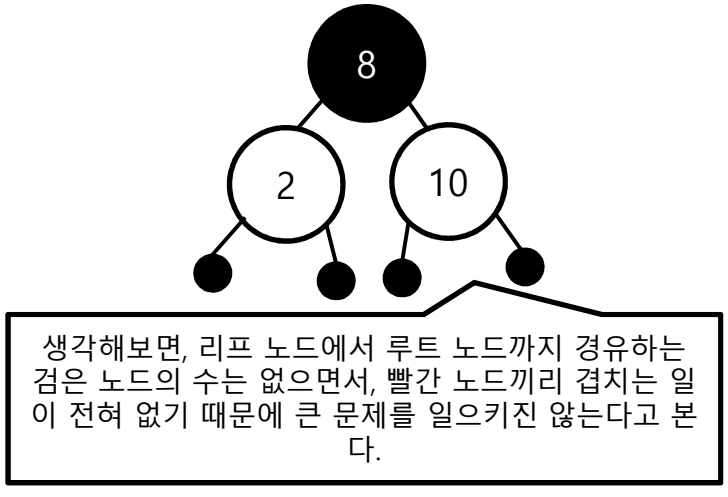
삽입하는 과정은 워낙 복잡하다고 생각할 수 있다. 그래서 이번에는 알고리즘을 살펴보면서 체계적으로 들어가보는 기회를 가져보자.

-> 삽입하는 노드는 언제나 빨간색으로부터 시작한다.

-> 삽입된 노드의 부모가 없다면, 바로 검은색으로 탈피를 한다.

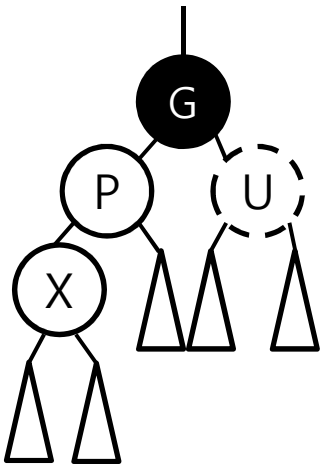


-> 삽입된 노드의 부모가 블랙이라면 레드블랙트리의 규칙에 대해서 어긋나지 않기 때문에 그냥 Pass한다.



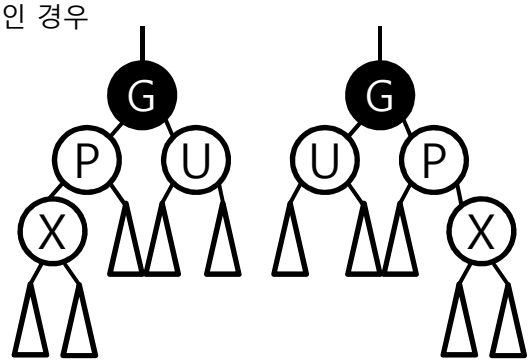
그러나 실질적인 문제는 지금부터이다. 바로 부모의 노드가 빨간색인 경우이다. 부모 노드가 빨간 노드이고, 자식 노드가 빨간 노드인 경우에는 레드블랙트리의 규칙에 어긋날 수 밖에 없기 때문에 이에 대해서는 색상 재조정 규칙이 필요하다. 이제부터가 좀 생각을 많이 해야겠다. 아래를 읽어보면서 이해를 해보자.

2-2) 삽입 후 색상 재조정 규칙
우선 부모의 노드는 빨강, 자식의 노드는 빨강이라는 점은 그대로 유지된다. 그러나 조부모 노드와 삼촌 노드에 대해서 어떻게 변하느냐에 따라서 색상이 재조정이 된다. 그래서 우리는 아래와 같은 트리를 사례로 설명에 들어가겠다.

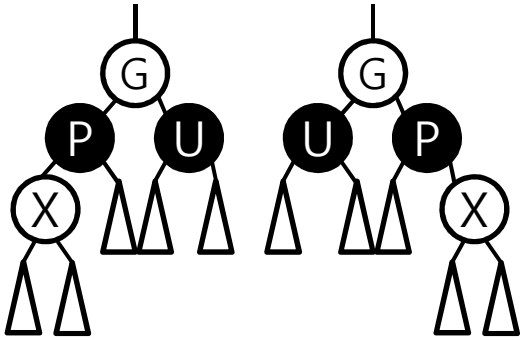


우선 X는 막 방금 삽입된 노드이다. P는 부모 노드(Parent). U는 부모와 같은 형제인 삼촌 노드(Uncle), G는 조부모 노드(GrandParent)이다. 여기서 삼촌 노드에 대해서 점선을 둔 이유가 삼촌의 노드에 따라서 부모의 노드와 X의 노드가 달라질 수 있기 때문이다. 여기서 이상한 삼각형 모양은 각 노드가 지니고 있는 서브트리라고 보면 되겠다. 트리의 모양은 삼각형처럼 생겼으니깐^^. 그리고 색상 재조정에서 제일 강조하고 싶은 점은 서브트리를 null 포인터가 아닌 방대한 트리로 보고 넘어가라는 것이다.
이에 대해서 4가지 경우를 나누어서 알아보도록 하겠다.

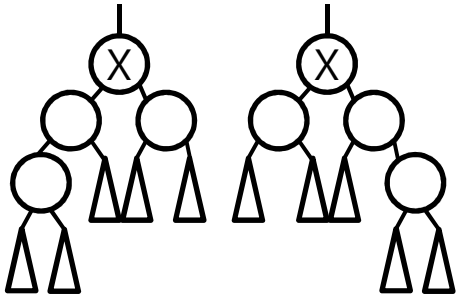
Case01) P노드와 X노드의 방향은 같은데 U 노드의 색상이 빨간색인 경우



일단 자식 노드가 X만 있는 것이 아니다. 그래서 양쪽에 공평성을 두기 위해서 P, U의 노드 색상을 검은색으로 변경을 해야겠다. 이러한 변경 작업을 통해서 루트노드부터 리프노드까지 검은색 노드를 즈려밟는 개수는 달라지지 않기 때문이다.

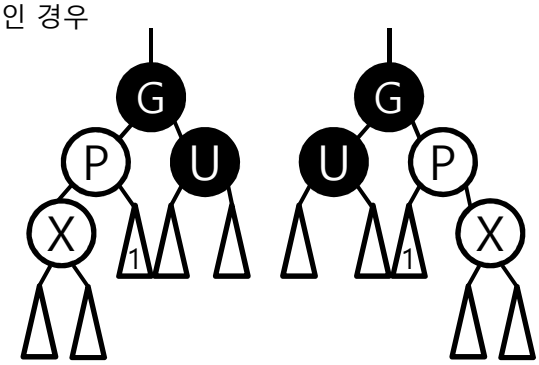


그렇지만 G노드에 대해서 이슈를 두게 되면, G노드의 부모 노드가 빨간색인 경우에 대해서 생각을 해볼 수 있겠다. 이에 대해서는 G노드에 대해서 재조정을 들어간다. 그리고 여기서 G노드는 루트 노드가 아닌 경우에 대해서 고려를 하면 복잡하게 되지만, G노드가 루트노드이면 그냥 검은색으로 칠하고 넘어가면 된다.

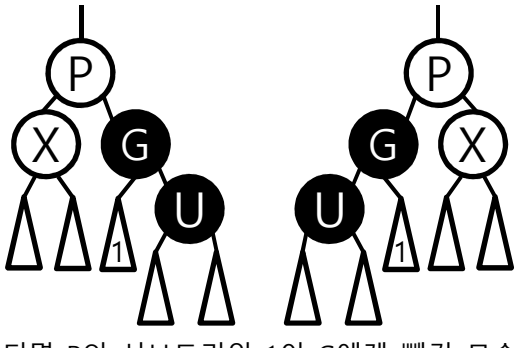


이제 마지막으로 X 노드에 대해서 색상 변경이 들어가게 되는데 G 노드를 X노드로 두고 적용을 하는 것으로 보면 되겠다. 분명 X노드 아래에도 자식 노드가 있으니 이에 대해서도 색상을 고려하는데 이는 재귀호출을 통해서 재조정을 한다. 여기서 재귀호출은 회전을 하는 방법도 있고, 여러 방안이 존재한다.

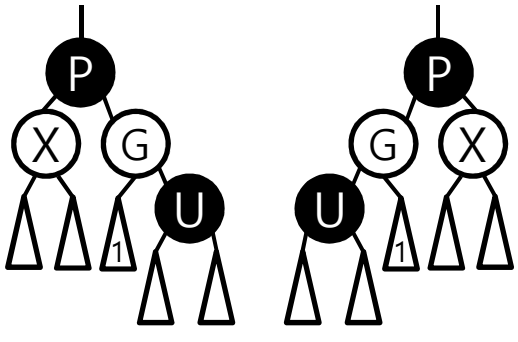
Case 02) P노드와 X노드의 방향은 같은데 U노드의 색상이 검은색인 경우



여기서 P를 검은색으로 칠하면 끝 아닌가요 라고 하는 사람들이 있지만, 아시다시피 X뿐만이 아니라 서브트리 1도 고려를 해줘야 하기 때문에 올바른 방안이 아니다. 그래서 우리는 각각 U 노드 방향으로 하나 꺾여 내려보겠다.

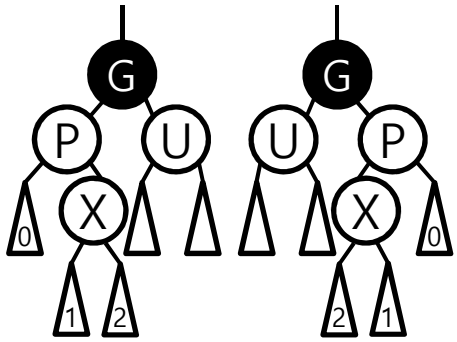


그렇게 되면 P의 서브트리인 1이 G에게 뺏긴 모습을 볼 수 있겠다. 이렇게 된 이유는 P는 분명 자식이 있게 되는데 공평성을 유지하기 위해서 어쩔 수 없이 G에게 서브트리를 주고 올라가야 P의 자식이 X와 G로 유지가 되기 때문이다. 이제 여기서 P와 G의 색깔을 서로 바꿔줌으로서 Case 02에 대한 경우는 상황 종료가 됨으로서 레드블랙트리를 만족하게 된다.

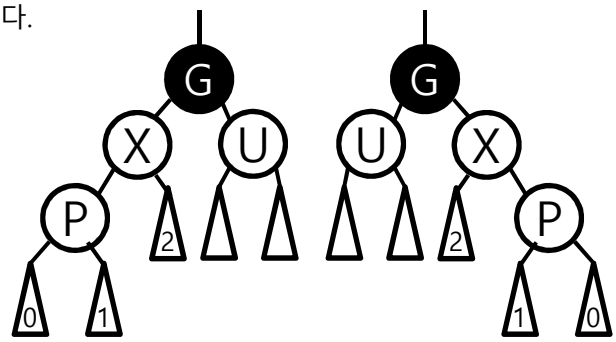


여기까지만 살펴봐도 자식 노드의 색깔은 빨간색으로 유지가 되지만, 부모 노드의 색깔은 검은색으로 변한다는 사실에 대해서도 짚어볼 수가 있다. 재조정을 하는 근본적인 이유가 바로 서로 빨간 노드인 경우에는 최근에 들어온 노드가 빨간색, 이전에 있는 노드는 검은색으로 형성함으로서 레드블랙트리의 조건에 대해 만족을 하는 원리로 생각할 수 있겠다.
계속해서 부모와 자식의 방향이 이번엔 서로 다른 경우에 대해 살펴보자.

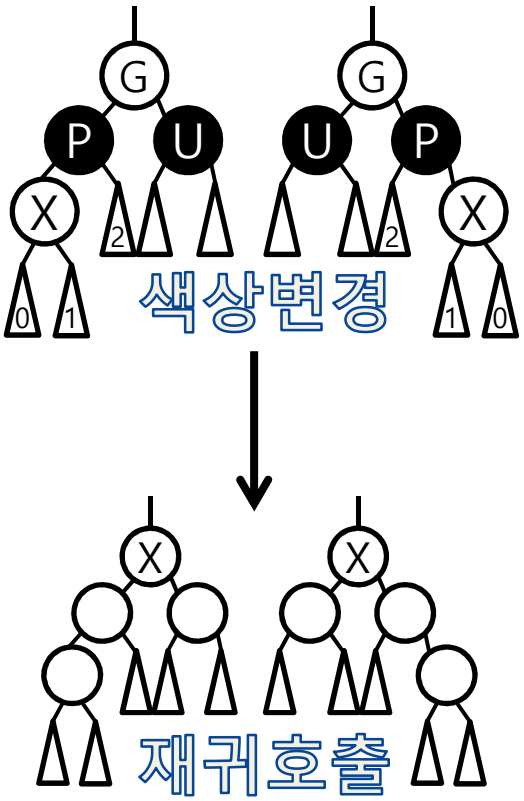
Case 03) P노드와 X노드 방향이 서로 다른데, U노드가 빨간색인 경우



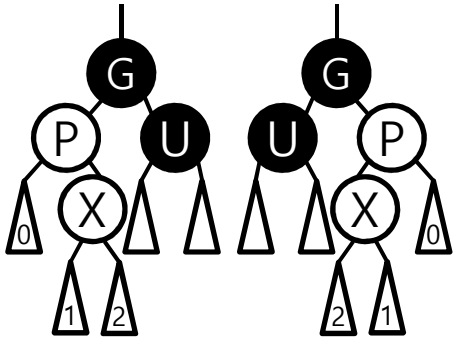
여기서는 P와 X의 레벨이 서로 바뀌어야 된다. 이처럼 되어 있는 경우에는 회전 작업이 이루어지는데 아래와 같이 바뀌겠다.



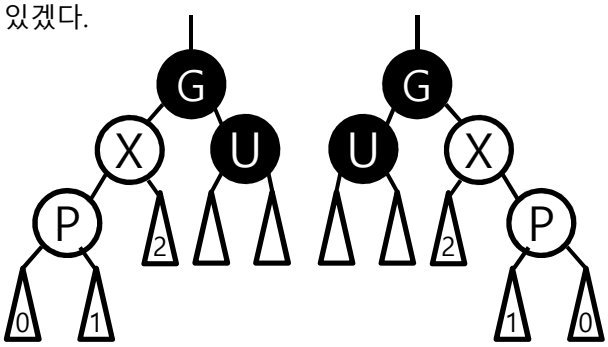
여기서 주목해야 하는 점은 X 노드의 1번 서브트리를 P 노드에게 주는 경우를 살펴볼 수 있다. 마치 Case 02번에서 P노드가 G노드에게 서브트리를 주는 원리와 같다고 볼 수 있다. 회전을 하게 되면 Case01과 같은 경우를 살펴볼 수 있다. 이렇게 되면 Case 01의 규칙에 따라서 아래와 같이 작업을 해주면 되겠다.(이에 대해서는 Case 01를 자세히 읽어볼 것.)



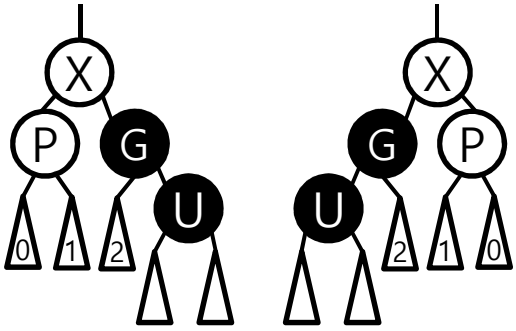
Case 04) P노드와 X노드의 방향이 다른데 U노드가 검은색인 경우



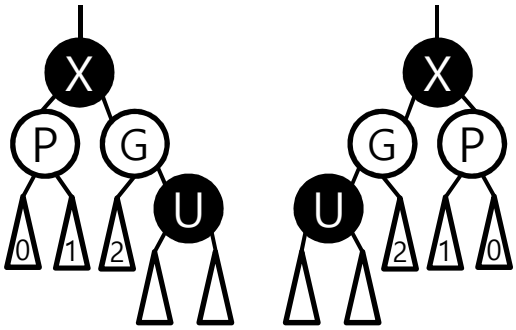
쉽게 이야기해서 Case 03과 마찬가지로 부모의 노드 방향과 자식의 노드의 방향이 다르면 우선 회전부터 시작을 하고 풀어 나가는 것이 상책이다. 그러면 아래와 같이 재구성을 할 수 있겠다.



이처럼 X의 서브트리를 P에게 주면서 Case 02에 대해서 실행을 하게 된다. 각자 U 노드 방향으로 꺾여 내려주면 아래와 같이 작성이 되겠다.



그러면 이처럼 어느 정도 과정을 거치고 나면 아까처럼 G 노드의 색상과 X노드의 색상을 변경을 해주면 레드블랙트리의 규칙에 어긋나지 않고 완성할 수 있게 된다.



2-3) 레드블랙트리 삽입의 사례

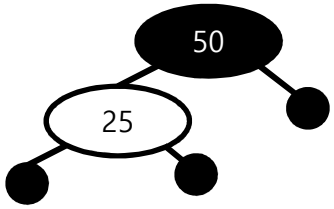
2-2를 보면서 이해를 하는데 있어서 고생이 많았다. 그렇지만 실질적으로 레드블랙트리가 삽입되는 사례를 살펴보게 되면 레드블랙트리를 접근하는데 있어서 더욱 쉽게 이해할 수 있겠다. 아래와 같은 데이터들을 삽입한다고 가정을 하겠다.

[50, 25, 15, 30, 65, 80, 20, 45, 23]

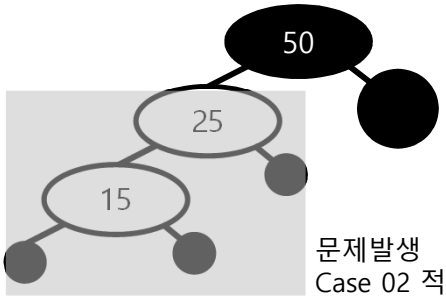
1단계) 50을 삽입한다면...

일단 노드를 삽입하면 루트 노드만 나오기 때문에 50만 삽입하고 끝난다.

2단계) 25를 삽입한다면...
25는 50보다 작기 때문에 언제나 늘 그랬듯이 왼쪽으로 간다. 여기까지만 해도 문제는 없다.

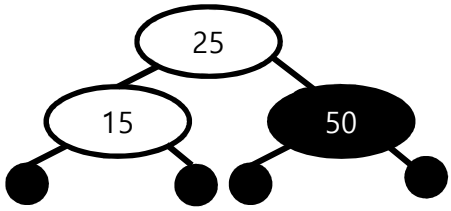


3단계) 15를 삽입하게 된다면...
이제부터 문제가 나왔다. 일단 모양을 그리면 아래와 같다.

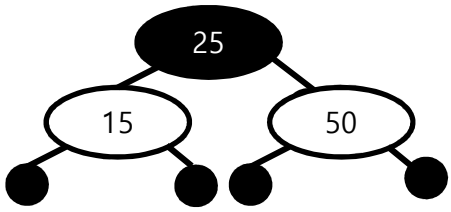


문제발생
Case 02 적용

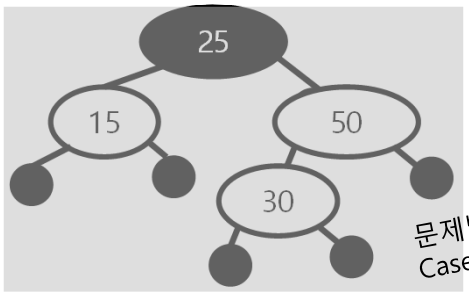
여기서 50의 null 포인터를 크게 그렸을까? 이는 U 노드라고 보면 되겠다. 부모와 자식의 방향은 같은데 삼촌 노드가 검은색인 경우에 대해 고려를 해야 하니 Case 02를 적용해야 하므로 아래와 같이 수정을 들어간다.



그러면 25와 50과 색상을 서로 바꿔주면 레드블랙트리에 있어서 문제 없이 종결된다.

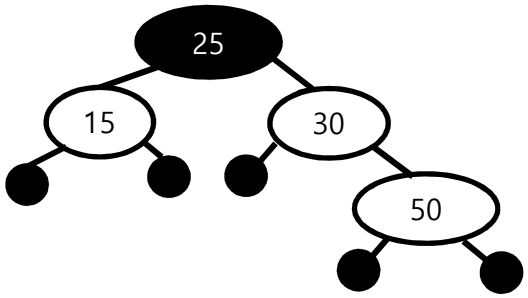


4단계) 30을 삽입하게 된다면...

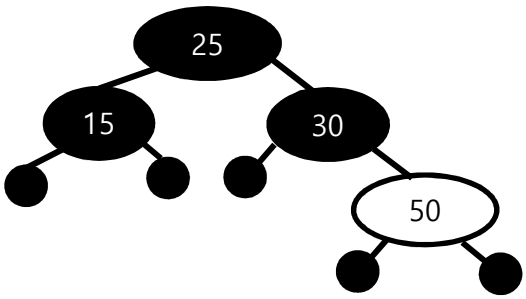


문제발생
Case 03 적용

30을 추가하면 이처럼 또 꼬여지기 때문에 전체의 트리를 통해서 문제를 해결해야 되겠다. 이는 Case 03에 대한 문제로 볼 수 있겠다. 30과 50을 회전하자.

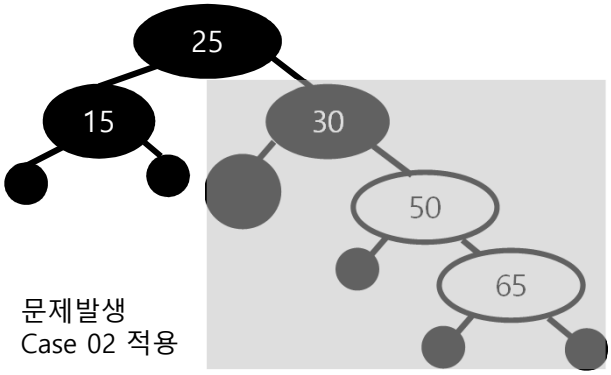


그러면 이번에는 Case 01과 같은 문제가 나오게 된다. 15와 30을 검은색으로 바꿔주면 25가 root인데 빨간색이면 안 되니 검은색으로 바꿔주고 종결한다.



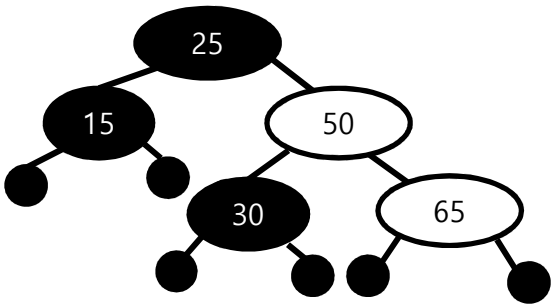
이처럼 색상을 변경하면 리프 노드에서 루프 노드까지 경로에서 블랙 노드를 경유하는 수는 같아지게 되면서 상황 종료.

5단계) 65를 삽입하게 된다면...

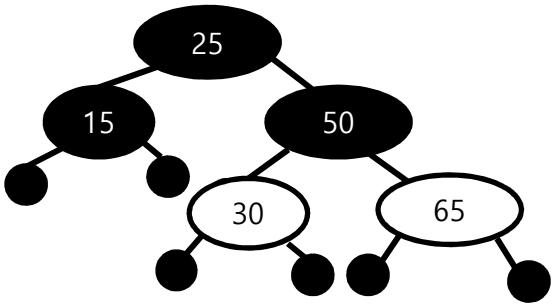


문제발생
Case 02 적용

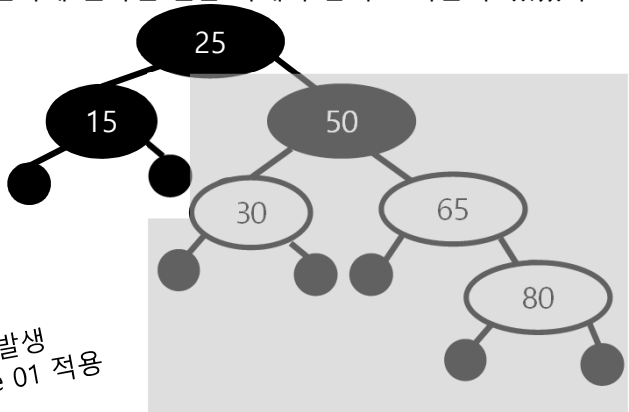
여기서 또 문제가 잡히게 된다. 이에 대해서는 Case 02와 같은 원리로 생각을 하면 되겠다. 구분하기 쉽게 하기 위해 큰 검은 동그라미는 삼촌 노드로 생각하면 되고 삼촌이랑 같이 낚시터에 놀러가는 것을 연장해서 삼촌 방향으로 꺾어 내려보면 아래와 같이 된다.



이제 여기서 50과 30의 색상을 바꿔주면 레드블랙 트리의 규칙에 벗어나지 않는 범위에서 트리가 완성된다.

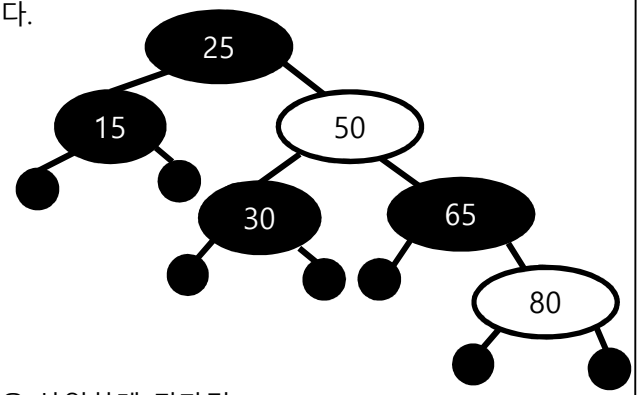


6단계) 80을 삽입하는 경우에는...
80을 삽입하게 된다면 일단 아래와 같이 그려질 수 있겠다.

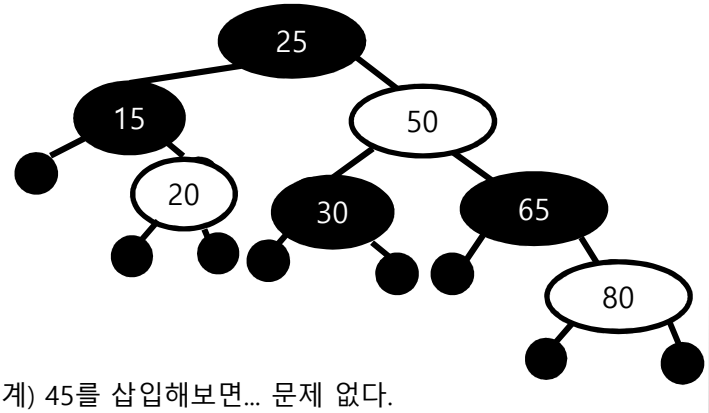


문제발생
Case 01 적용

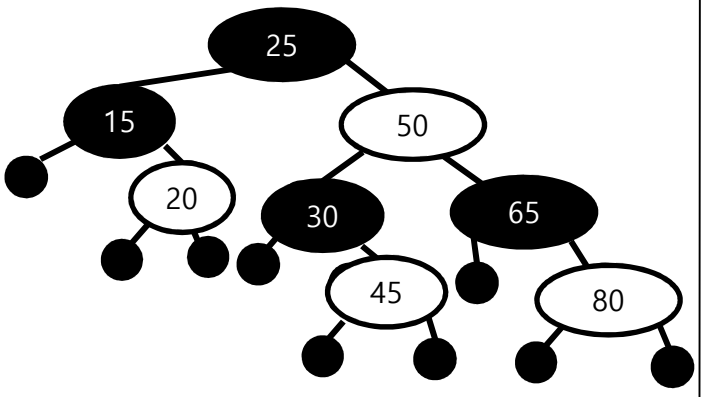
여기서 30과 65의 색을 검은색으로 바꾸고, 50을 빨간색으로 바꿔주면 비로소 레드블랙트리의 규칙을 지키는 범위에서 완성된다.



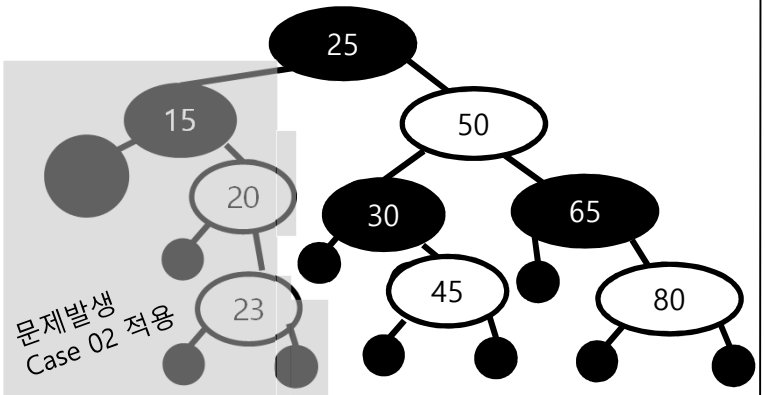
7단계) 20을 삽입하게 된다면...
그렇다... 20을 삽입하는데 큰 문제가 없다.



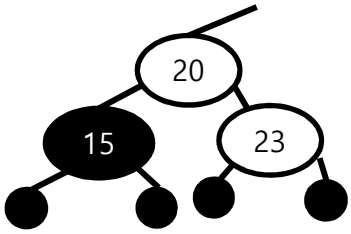
8단계) 45를 삽입해보면... 문제 없다.



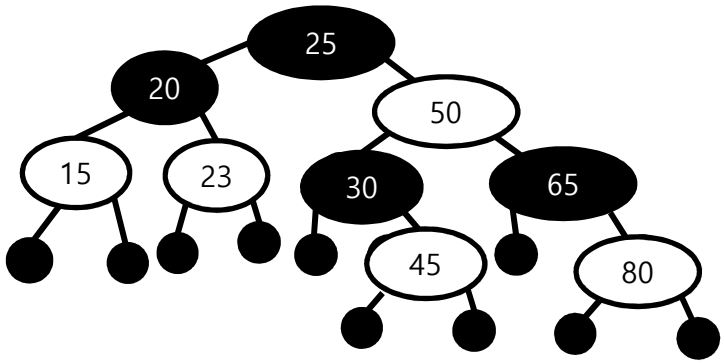
9단계) 23을 삽입하게 된다면...



어우... 생각보다 데이터들이 많이 들어있다. 이 경우는 Case 02에 대해서 오류가 나기 때문에 삼촌 방향으로 낚시대를 꺾어 내려보도록 하겠다.

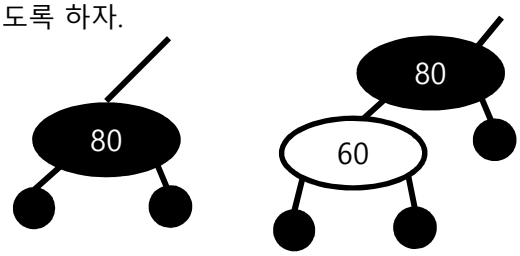


이제 이 부분에서 15의 색상과 20의 색상에 대해서 변경을 해주게 되면 레드블랙 트리에 대한 조건을 모두 만족하게 되어 완성이 된다.



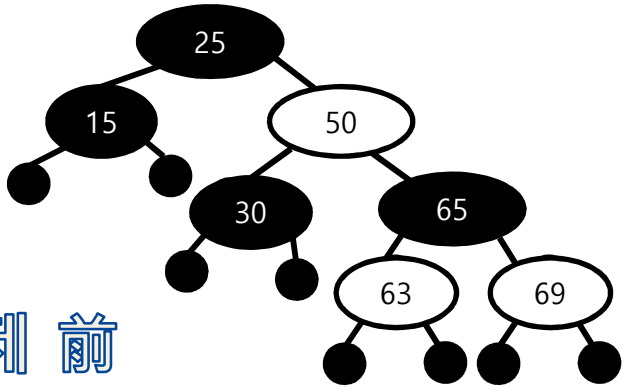
3. 레드블랙트리의 삭제 과정

레드블랙트리의 삽입 과정까지만 해도 null 값에 대해서도 노드로 취급을 하기 때문에 null 노드가 아닌 노드들은 전부 (즉 마지막에 새로이 추가된 노드들까지도) 자식을 2개 가지고 있다고 보면 레드블랙트리 개념에 대해 접근할 수 있었다. 그러나 삭제 과정에 대해서 자식의 수에 대해서 null 노드를 예외하고 넘어가야겠다. 자식이 하나인 노드는 null 노드를 떠나서 자식이 하나라는 뜻이다. 아래와 같은 노드를 살펴보도록 하자.



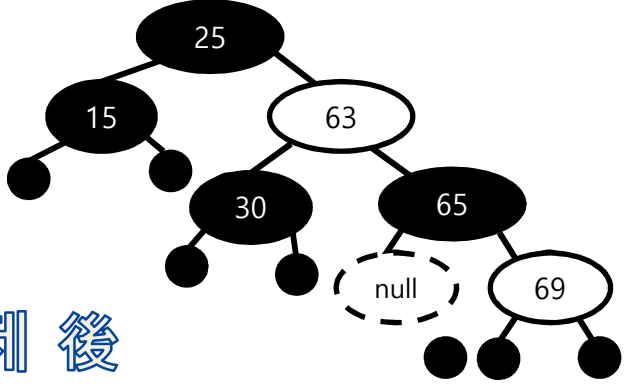
이처럼 왼쪽의 80과 같은 노드는 그 뒤에 자식들이 null 밖에 없기 때문에 이는 자식이 없다. 그렇지만 60이란 값을 추가하게 된다면 80이란 노드는 자식이 한 명인 것으로 판단될 수 있다. 이처럼 자식들을 기준으로 해서 레드블랙트리의 삭제 과정은 어떻게 이루어지는지에 대해 알아보도록 하자.

3-1) 자식이 2개인 노드를 삭제하는 경우에는...



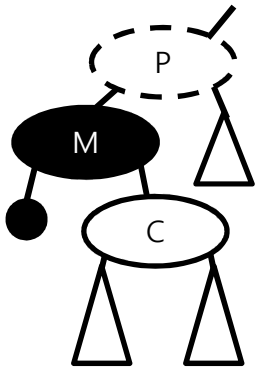
삭제 前

자식이 2개인 노드에 대해서 삭제하는 과정에 대해서는 색깔에 대해서 고민할 필요가 없다. 노드 50을 삭제한다고 가정을 한다면 50의 우측 서브트리에 대해 getLeftMostValue를 통해서 최솟값인 63으로 바꿔야 삭제 과정이 진행된다. 그렇지만 50이랑 63은 색상이 같더라도, 아니 다르더라도 그냥 50은 63으로 바꾸고, 우측 서브트리에서 63을 삭제하는 과정으로 종결하면 된다. 이러한 과정을 통해서 오른쪽 서브트리에서 최솟값 노드에 대해 왼쪽 자식이 null이어야겠다.

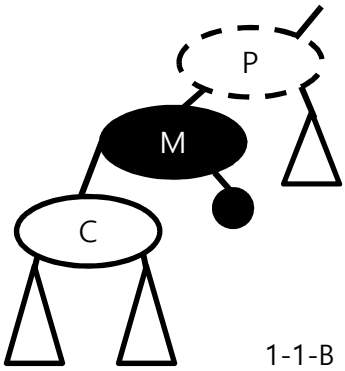


삭제 後

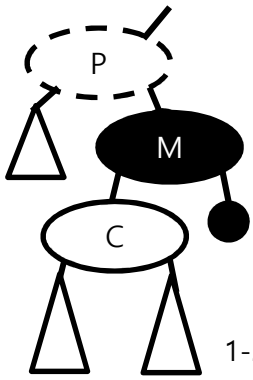
3-2) 자식이 한 개인 노드에 대한 삭제
삭제하려는 노드의 조합으로서는 3가지로 나뉠 수가 있겠다.
-> 블랙 / 레드
블랙/레드 조합은 삭제하려는 노드가 검은색이고, 자식이 빨간색인 경우를 살펴볼 수 있겠다.



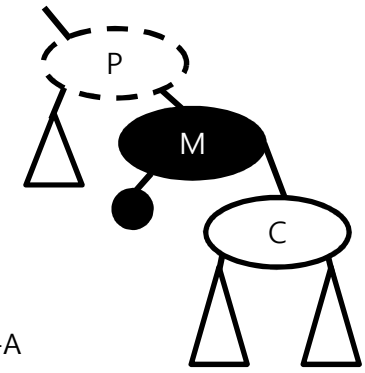
1-1-A



1-1-B

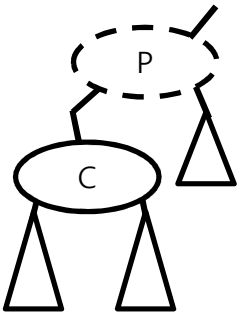


1-2-A

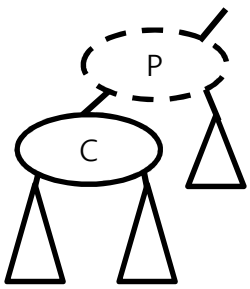


1-2-B

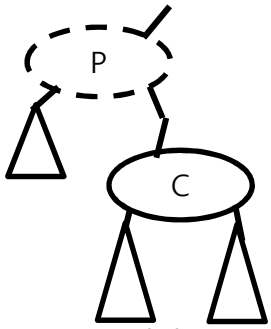
여기서 삭제하려는 노드가 M이고, P는 삭제하려는 노드의 부모, C는 삭제하려는 노드의 자식이라고 볼 수 있겠다. 일단 P 노드의 색상에 대해서는 생각을 하지 말고 넘어가보도록 하자. 그러면 이처럼 1-1-A, B 트리에 대해서는 M이 좌측 서브트리에 있는 루트 노드이고, C는 각각 M의 우측, 좌측 자식에 있는 경우로 살펴볼 수 있겠다. 그리고 1-2-A, B에 대해서는 M이 우측 서브트리에 있는 루트 노드이고, C는 각각 M의 좌측, 우측 자식노드로 볼 수 있겠다. 그럼 M을 삭제하게 된다면 어떻게 결과가 나올까? M을 없애고 보면...



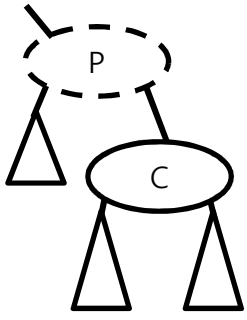
1-1-A-M제거



1-1-B-M제거

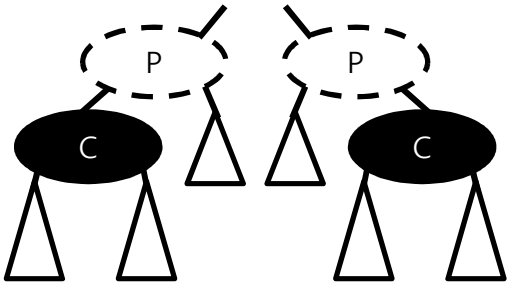


1-2-A-M제거



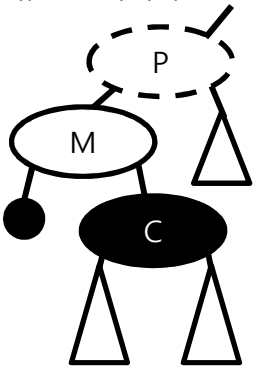
1-2-B-M제거

그러면 1-1-A,1-2-A에 대해서 M을 제거하게 된다면 C노드에 대해서는 결국 P에만 연결이 된다. 그리고 1-1-B, 1-2-B에 대해서 M을 제거하게 된다면 C노드에 대해선 결국 P에 그대로 연결된다. 그럼 각각 1-1-결과, 1-2-결과를 통해서 레드블랙트리를 유지하기 위해서 C의 색상을 바꿔주면 되겠다.

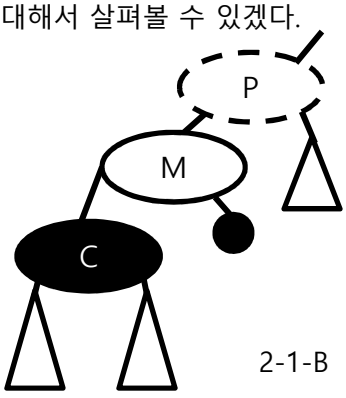


1-1-결과 1-2-결과

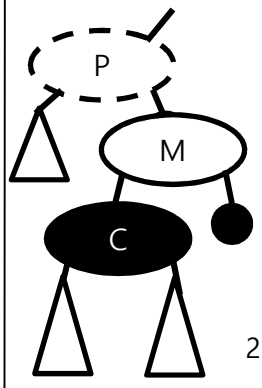
이처럼 삭제를 완료하고 난 뒤에 C의 색상을 검은색으로 바꾸게 된다면 경로상 블랙노드의 수에 대해서는 그대로 유지하게 된다. 일단 아까 삽입 알고리즘처럼 이러한 유형이 있다는 점에 대해서 알고 넘어가면 좋겠다.
-> 레드 / 블랙
레드/블랙 조합은 삭제하려는 노드가 빨간색이고, 그 녀석의 유일한 자식이 검은색인 경우에 대해서 살펴볼 수 있겠다.



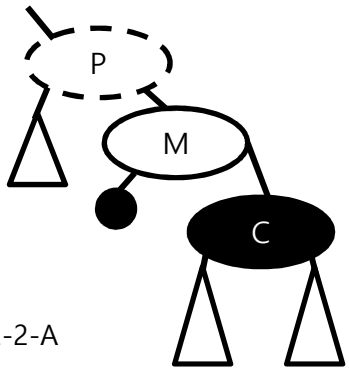
2-1-A



2-1-B

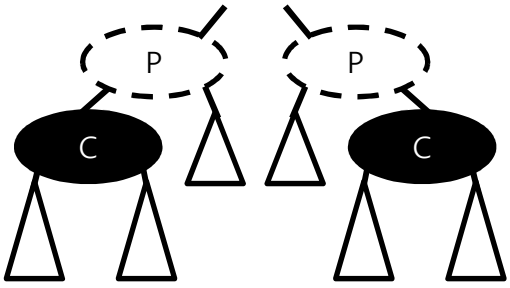


2-2-A



2-2-B

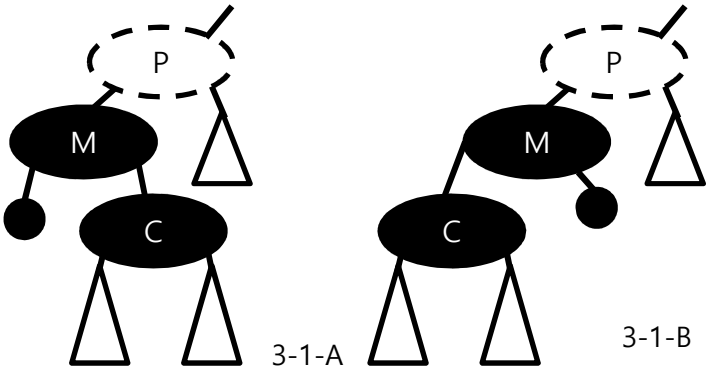
여기서는 간단하게 끝난다. 오로지 M이란 녀석에 대해서 삭제를 한다고 가정을 한다면 P와 C끼리 다시 연결함으로서 레드블랙트리를 유지하는 점에 대하여 결과가 끝난다. 이런 사례에서는 색상 변경에 대해서 들어가지 않고 P와 C를 연결하는 점에서 종결이 되겠다.



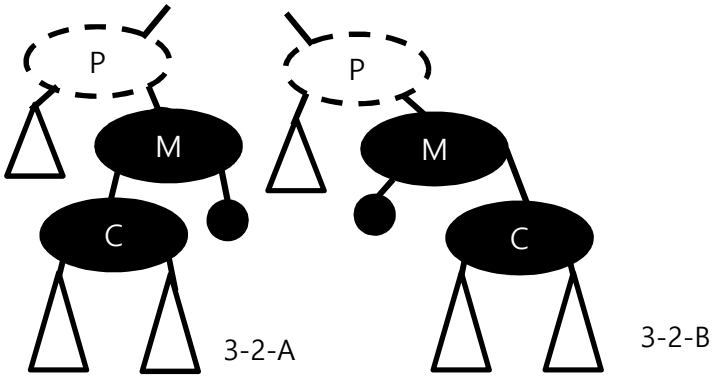
2-1-결과 2-2-결과

-> 블랙 / 블랙
삭제하려는 노드가 검은색이고, 그 노드의 자식이 검은색인 경우에 대해서는 검은색 노드가 레드블랙트리 규율에 영향이 크기 때문에 삭제를 잘 해줘야겠다. 그렇지만 이 과정에서 제일 중요한 것은 만의 하나를 통해서 M 노드를 삭제하고 난 후에는 재조정 규칙이 필요로 하다. 이에 대해서는 다음 쪽에서 살펴보도록 하자.

-> Next Page ->



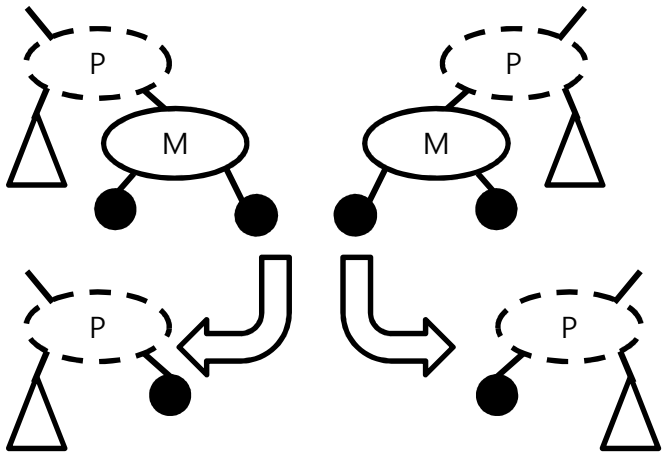
3-1-A 3-1-B



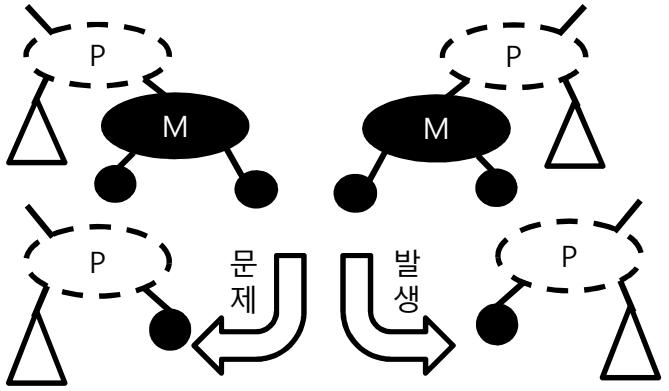
3-2-A 3-2-B

여기서 M노드를 삭제를 하게 된다면 분명 C를 연결한다는 점에 대해선 말 안 해도 알 것이다.(현재 3-1-A부터 3-2-B까지 레드블랙트리의 규율의 4번째는 만족한다고 가정을 한다.) 그렇지만 여기서 중요한 것은 모든 경로 상에서 블랙 노드 경유의 수가 같아야 되는데 C노드로부터 루트까지 올리는데 블랙 노드가 하나 부족하다. 그래서 삭제 후 트리 재조정에 대해서 알아볼 필요가 있다... 이에 대해서는 경우의 수가 워낙 많은데 그래도 오픈북인데도 불구하도 어떠한 원리로 돌아가는지에 대해서 공부를 해둘 필요가 있다. 이는 자식이 없는 노드의 삭제 과정 이후에 살펴보도록 하겠다.

3-3) 자식이 없는 노드의 삭제과정

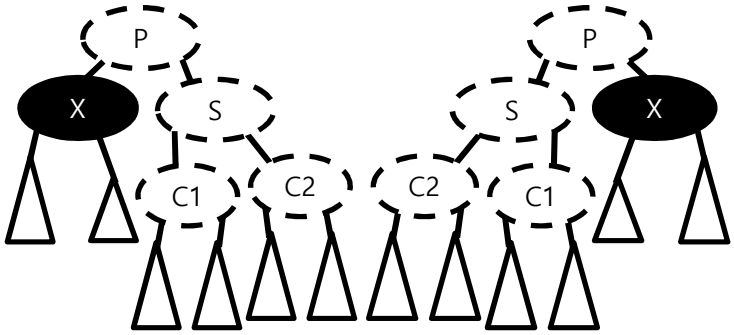


여기서 M노드를 삭제를 하게 된다면 null 노드에 대해서 P에 자식으로 연결을 한다. 레드 노드에 대해서 삭제하는 경우에 큰 문제는 발생하지 않음으로서 종결이 된다. 하지만 M이 블랙 노드이면 말이 달라지겠다.



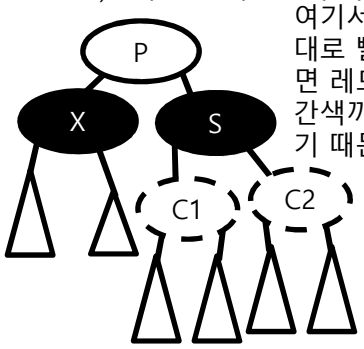
방금 전처럼 현재까지 레드블랙트리의 규칙을 지키는 범위 이내에서 그 뒤에 블랙 노드를 삭제하게 된다면 블랙 노드가 하나 부족하게 되면서 트리 재조정 규칙이 필요하게 된다. 이에 대해서 자세히 공부를 해볼 필요가 있겠다.
이제부터 다양한 트리의 사례가 소개가 될 것이니 여기서부터 집중을 해주길 바란다.

3-4) 삭제 후 트리 재조정 규칙
아까와 같이 삭제 3-2의 마지막 부분과 3-3과 같은 경우를 살펴본다면 분명 삭제하는 노드 때문에 규칙이 깨졌으니 블랙 노드가 책임을 져야 할 수 밖에 없는 상황이다. 그래서 이에 대해서 재조정을 하는 방법에 대해서 소개를 할 것이다. 그림이 많이 나오니깐 암기를 하는 목적이 아니라 이해를 하는 목적으로 보면 도움이 되겠다. 어자피 오픈북 시험이니깐 이해를 하는 마음을 가지도록 하자.



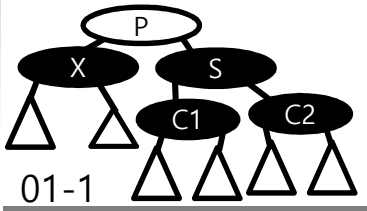
여기서 X노드는 현재 삭제 노드 M때문에 규칙이 깨졌으니 책임을 져야 하는 노드이다. 그럼 여기서 P는 당연히 M의 부모로 보면 되겠고, S는 sibling이란 약자로서 형제라는 뜻이다. 뭐 brother라는 말도 있긴 한데 여기서는 sibling이라고 하면 그런 줄 알고 넘어가자. 여기서 C1이랑 C2는 당연히 S의 자식 노드인 건 아는 사실이다. 그렇지만 X 노드와의 차이를 살펴보면 C1은 X노드와 가까운 쪽으로 보면 되겠고, C2는 X노드와 먼 쪽으로 보면 되겠다. 이에 대해서 왜 구분을 하는 것일까? 바로 P라는 녀석이 꺾어 내려간 경우에 우측 서브트리(혹은 좌측 서브트리)로 연결이 되기 위해서 구분할 필요가 생긴 것이다. 이에 대해서 색으로 구분을 하면서 알아보도록 하자.

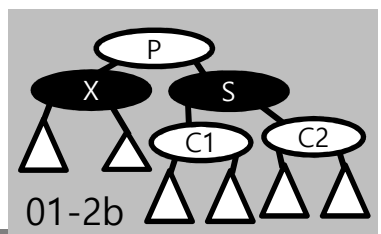
일단 P라는 노드에 대해서 색깔을 입혀보면서 차이점을 알아보도록 하겠다.
Case 01) P가 빨간색인 경우에는

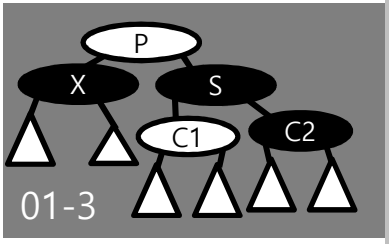


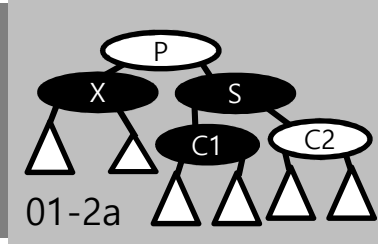
여기서 살펴봐야 되는 것은 S는 절대로 빨간색이 될 수가 없겠다. 왜냐면 레드블랙트리의 원칙 상에서 빨간색끼리 연속으로 존재할 수가 없기 때문이다.

그래서 C1, C2에 대해서 빨간색과 검은색으로 2x2=4가지 경우의 수로 나뉘어서 살펴볼 수 있겠다. 색상이 진해질수록 회전 과정이 많아지겠다...

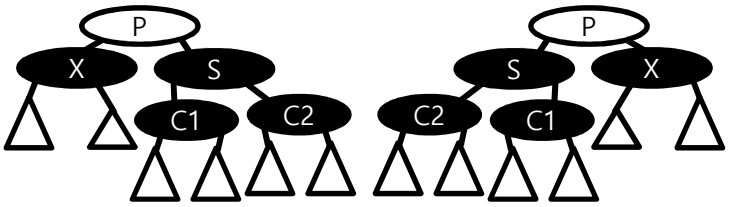




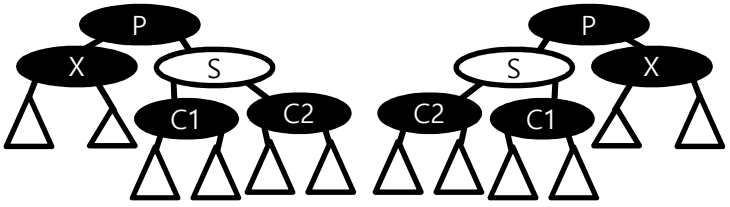




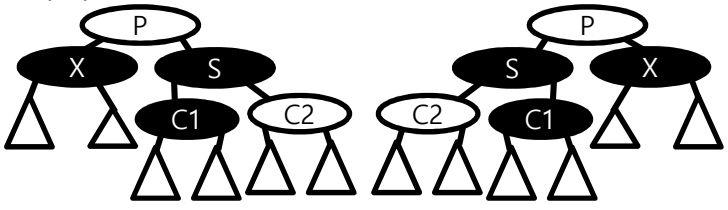
01-1) 부모 노드가 레드인 대신에 모든 노드가 블랙인 경우에



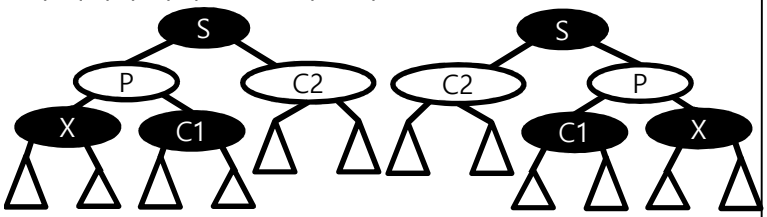
여기서 해야 하는 것은 회전이 아니다. 오로지 P랑 S랑 색깔을 바꿔주면 블랙 노드의 경유 수가 같아지기 때문에 간단하게 해결할 수 있겠다.



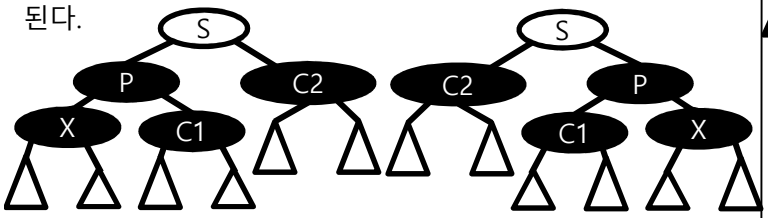
01-2a) X노드랑 가까운 노드가 블랙이고, 먼 노드가 레드인 경우에는...



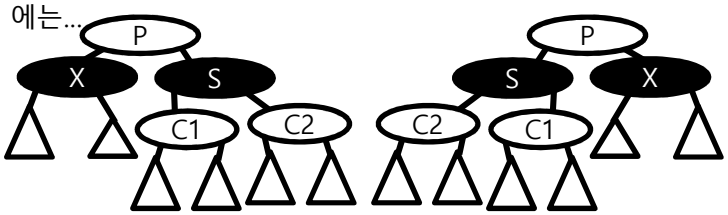
우선 S노드를 위로 올리면서 노드 X가 책임지기 위해서 그 부분으로 방향을 꺾어줘야겠다. 그럼 C1은 자연스레 P노드에 대해서 자식으로 남게 된다.



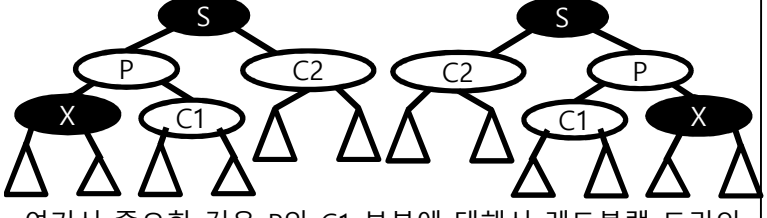
그럼 이처럼 작업이 끝나게 된다면 S와 그 자식들에 대해서 색상 변경에 들어가게 된다면 규칙을 유지하면서 종결하게 된다.



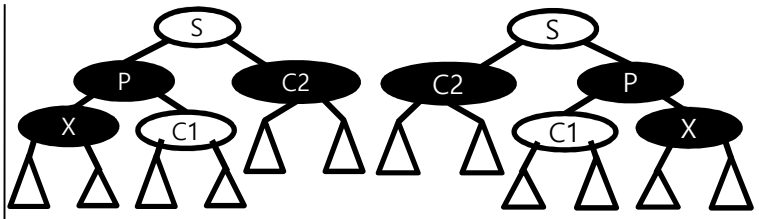
01-2b) X노드랑 가까운 노드든 먼 노드든 모두 레드인 경우에는...



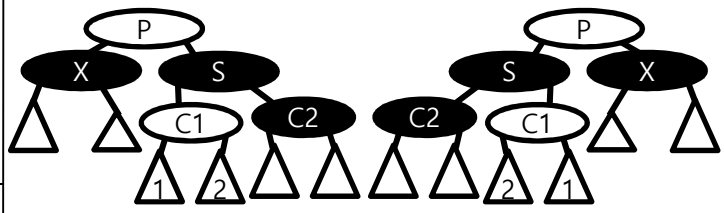
아까와 1-2a와 같은 원리를 이용해서 돌아가니깐 이에 대해서는 아까처럼 X방향으로 꺾어주도록 하자.



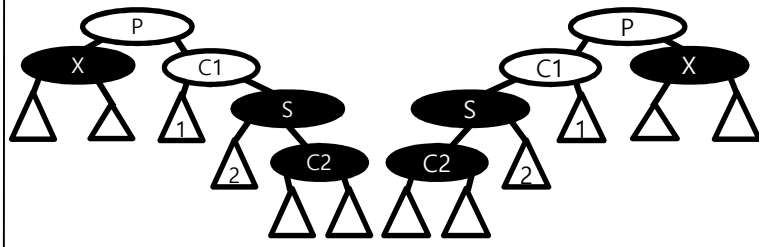
여기서 중요한 것은 P와 C1 부분에 대해서 레드블랙 트리의 규칙을 어길 수 밖에 없다. 그래서 C1에 대해서는 빨간색으로 그대로 두고, 아까처럼 S 노드와 그 자식들에 대해서 각자의 색상을 변경해주면 레드블랙트리의 규칙을 지켜낼 수 있겠다.



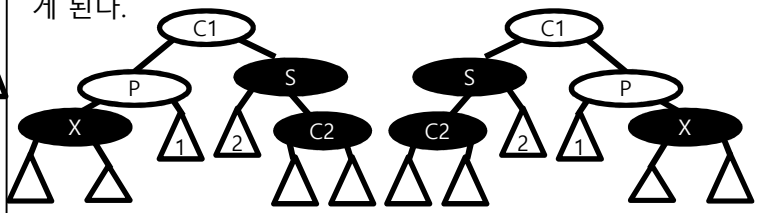
01-3) X노드랑 가까운 노드가 블랙이고, 먼 노드가 블랙인 경우에...
아까 01-2) 부분에서 살펴본 거와 달리 꽤나 복잡하고 어려울 수 밖에 없다. 이에 대해서 어떻게 돌아가는지에 대해서 그림으로 표현을 해보겠다.



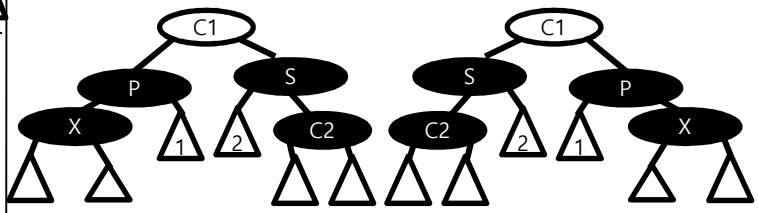
우선 C1에 대해서 S에 꼽사리를 시켜둬야겠다. 어자피 S라는 녀석은 이 과정에서 다시 올라가게 되기 때문이다.



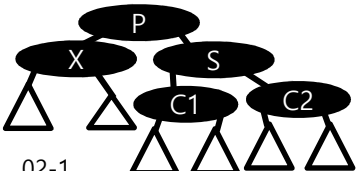
그럼 C2에 대해서 레드블랙트리 규칙에 어긋나게 되기 때문에 X노드가 책임을 져야 된다. 그래서 X노드 방향으로 아래로 꺾어 버리겠다. 또한 여기서 C1의 좌측 서브트리는 그대로 가지되, C1의 우측 서브트리(2번 섭트리)는 S에게 물려주게 된다.



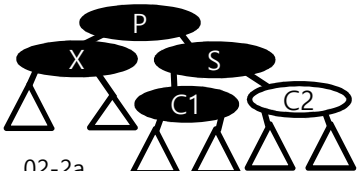
또 꺾는 과정에서 1번 서브트리는 P에게 물려주게 되면서 C1가 올라가게 된다. 그래서 이러한 과정을 통해서 C1의 색깔은 유지하게 됨으로서 P의 색깔을 검은색으로 바뀌게 되어 레드블랙트리의 과정에 만족을 하게 된다.



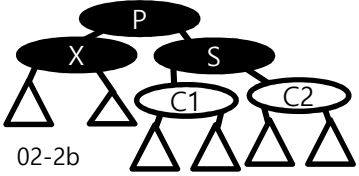
Case 02) P가 검은색인 경우에는...
일단 모든 노드가 검은색인 경우에는 레드블랙트리 규칙에 맞춰져있다고 가정을 하고 넘어가야겠다. 그러면 이렇게 4가지의 경우에 대해 살펴볼 수 있겠다.



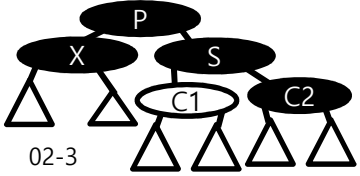
02-1



02-2a

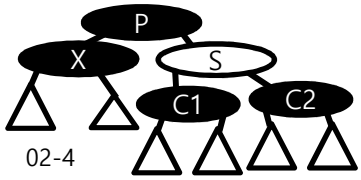


02-2b



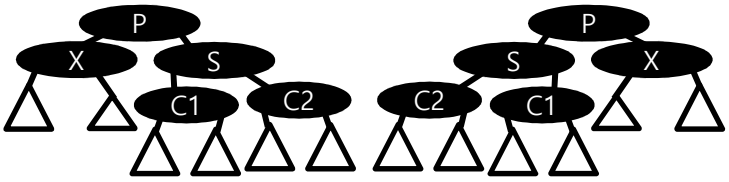
02-3

그렇지만 형제 노드가 빨간색인 경우에는 한 가지 경우밖에 안 나온다. 왜냐면 형제 노드들의 자식들이 어느 하나라도 빨간 노드인 경우에는 레드블랙 트리에 대해서 규칙을 만족할 수가 없기 때문이다. 그럼 이제 아까처럼 복잡하지만 열심히 살펴보도록 하자.

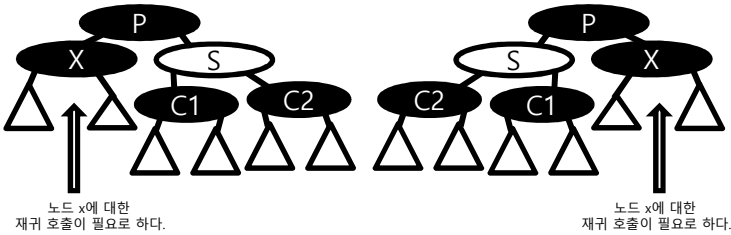


02-4

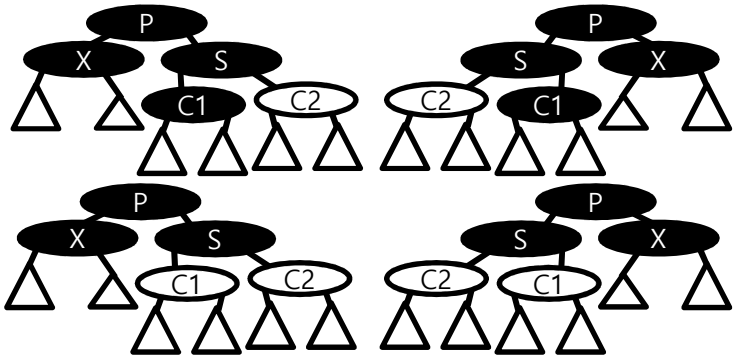
02-1) 모든 노드들이 검은색인 경우에...



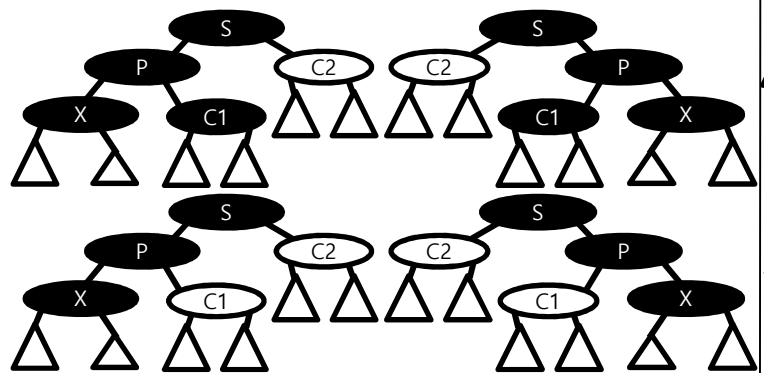
이에 대해서는 형제 노드에 대해서 빨간색으로 변경을 해주면 모든 리프 트리에서 루트 노드까지 검은 노드 경유의 수가 형평성있게 조절이 된다. 그렇지만 X에 대해서 재귀 호출을 통해서 구현을 해야 되겠다. 이는 회전을 하든 자리를 바꾸든에 대해서도 포함이 되겠다.



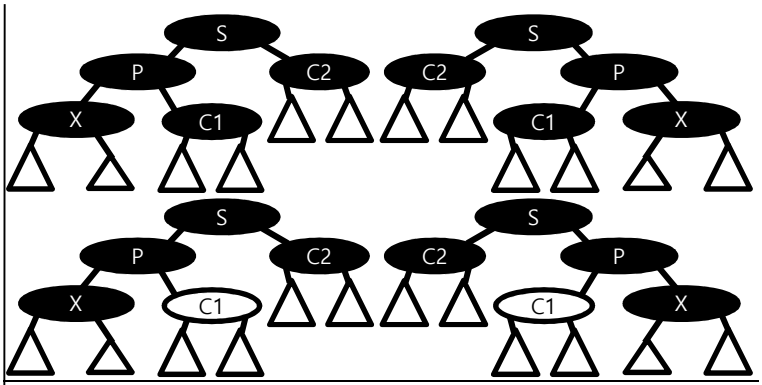
02-2a, b) X노드와 거리가 먼 자식 노드가 빨간색인 경우랑 자식들이 모두 빨간색인 경우에



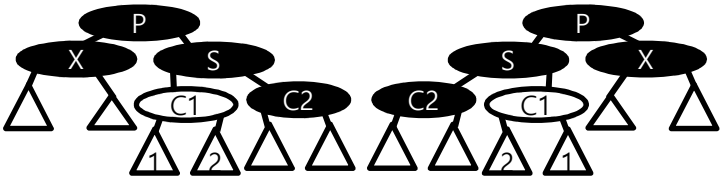
이 원리는 아까처럼 비스무리하니 묶어서 설명하는 것이 더욱 낫겠다. 우선 X에 대해서 책임을 져야 하니 모두 X노드에 대해서 내려보도록 하겠다.



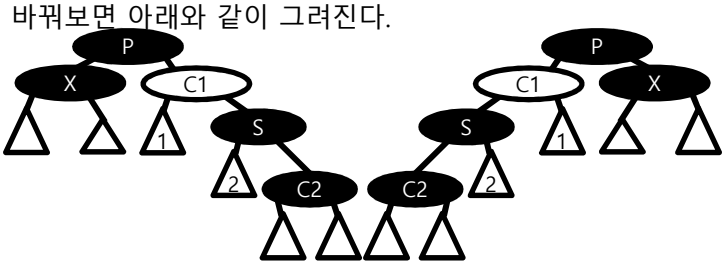
그러면 여기서는 달라지는 것이 바로 형제 노드의 직속 자식들 중에서 빨간색을 없애라는 점이 바로 포인트이다. 이를 적용하게 된다면 C2에 검은색만 칠하면 끝이다. 그러면 최종적으로 레드블랙트리가 정리가 된다면 오른쪽 위와 같이 정리가 될 수 있겠다.



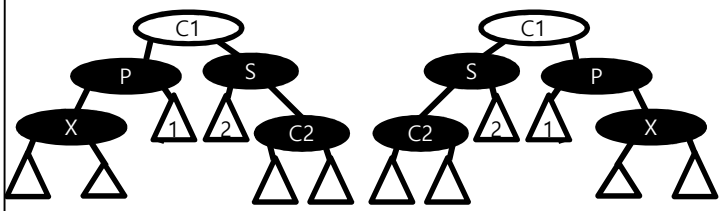
02-3) X노드와 거리가 가까운 자식 노드가 빨간색인 경우에



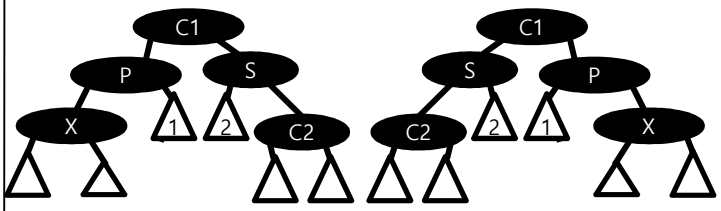
아까 01-3번과 같이 본다면 이제 요령이 생길 것이다. C1이란 녀석을 일단 S의 직속 자식으로 두고 난 후에 01-3번처럼 바꿔보면 아래와 같이 그려진다.



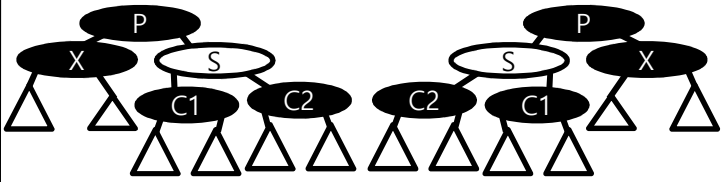
아까처럼 C1의 좌측 서브트리는 그대로 가지고 있고, C1의 우측 서브트리(2번 서브트리)는 S에게 물려준 모습을 볼 수 있을 것이다. 이처럼 작업을 하면 X방향으로 꺾임으로서 책임을 물고 넘어가면 되겠다.



이제 여기서 떡하니 보면 C1 하나만 검은색으로 바꿔주면 레드블랙트리의 규칙 중 1번째를 지킴으로서 이에 대해 준수하게 된다. Root는 항상 검은색이어야 되기 때문이다.

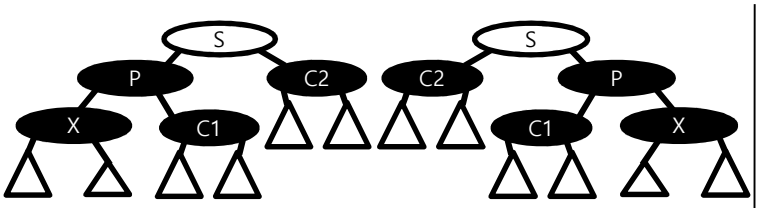


02-4) 형제 노드만 빨간색인 경우에...

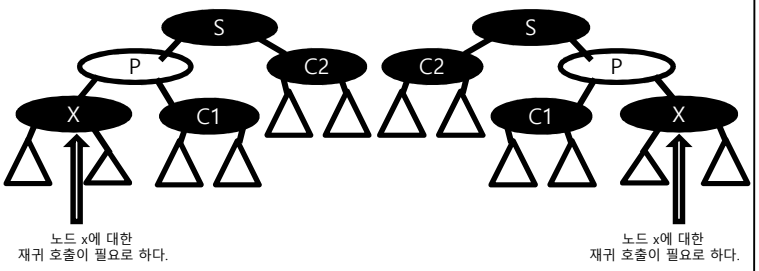


일단 형제 노드가 빨간색이라면 여기서는 노드 X 방향으로 꺾어주고 난 뒤에 봐야 된다. 그렇지만 여기서 중요한 것은 이러한 작업 이후에도 색변환에 재귀호출을 해야 한다는 것이다...

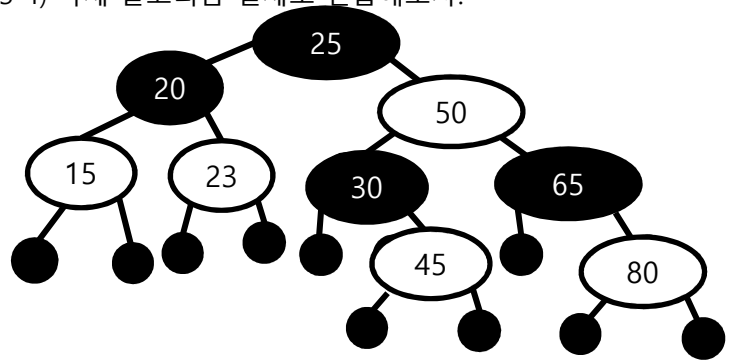
-> Next Page ->



그러면 여기서 P와 S 노드의 색깔만 서로 바꿔주는 역할을 함으로서 종결을 하면 되지만, X 노드에 대해서는 아직 해결되지 않는 경우가 있다. 이는 재귀 호출을 통해서 해결을 하면 되겠다.

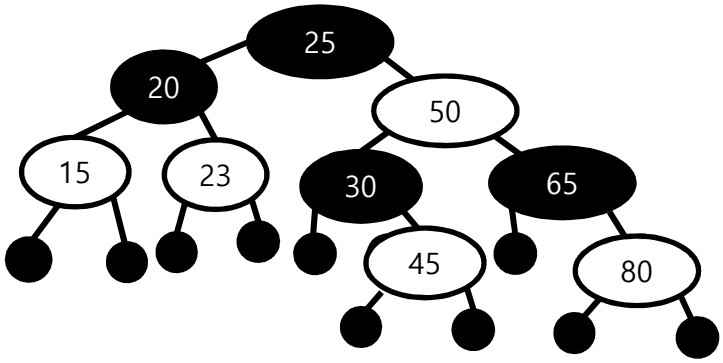


3-4) 삭제 알고리즘 실제로 연습해보자.

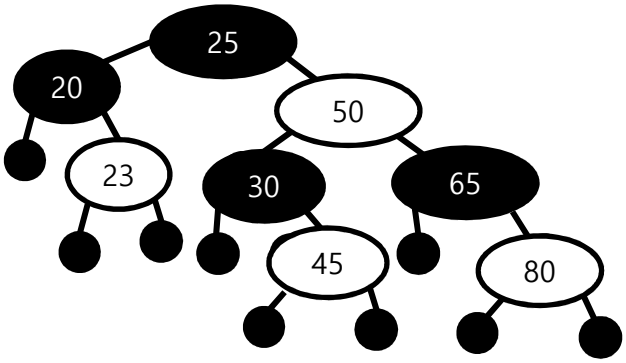


아까와 같은 순서대로 삭제를 해보는 연습을 한 번 해봐야겠다. 일단 순서는 아래와 같이 볼 수 있고, 이에 대해선 굳이 암기로 하는 것이 아니라 이해를 하면서 하는 관점으로 봐야 된다. 다시 한번 말하지만 실제로 공부를 해보면서 여러 번 연습 해보길 권장한다.

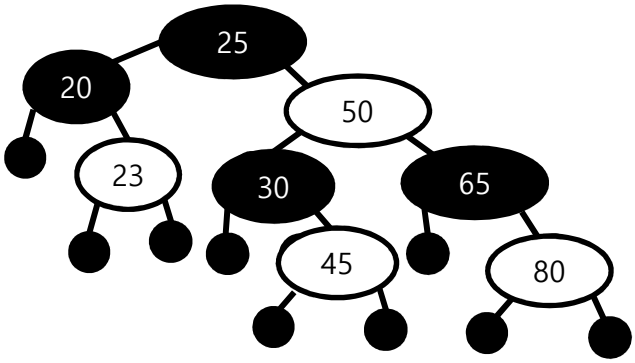
1단계) 15를 삭제한다면...



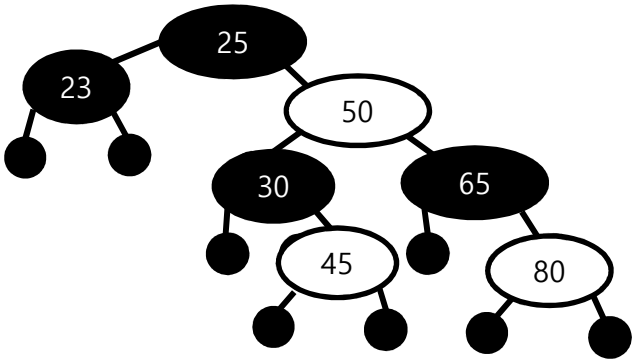
일단 15의 노드 색깔은 빨간색이다. 그렇지만 이에 대해서는 굳이 큰 영향을 주지 않기 때문에 그냥 삭제하면 되겠다.



2단계) 20을 삭제한다면...

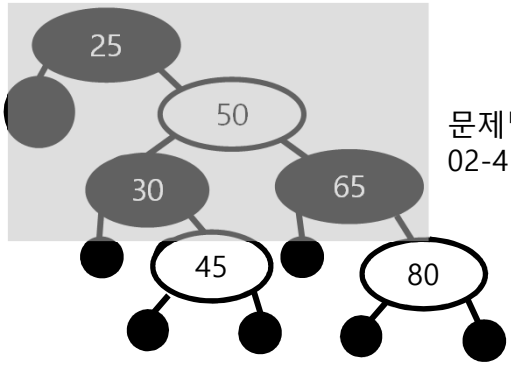


여기서 20을 삭제하는 과정을 살펴보면 23이란 녀석이 분명 올라오게 되면서 검은색으로 색깔이 바뀌게 된다.



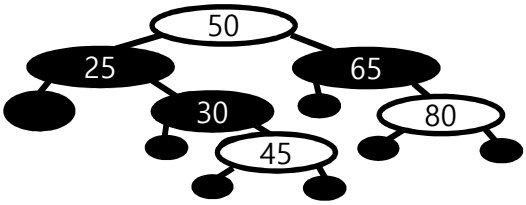
그런데 23이 혼자 있더라도 이 규칙에 대해선 깨지지 않게 되었다. 왜냐면 각각 리프 노드로부터 루트 노드까지 경유하는 검은색 노드의 수는 같기 때문이다.

3단계) 23을 삭제하게 된다면...



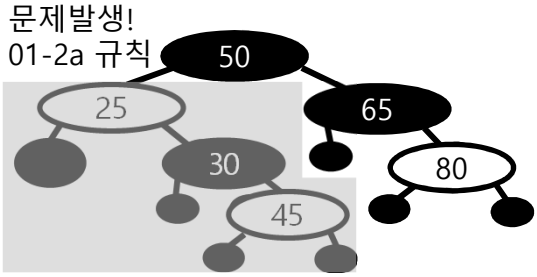
문제발생!
02-4 규칙

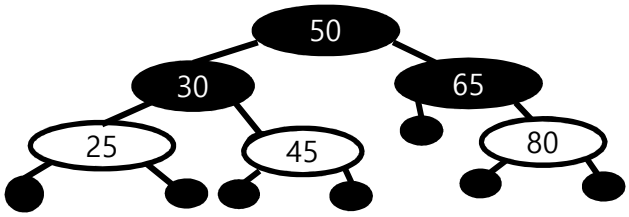
검은색끼리 삭제를 하면 분명 문제가 이제부터 시작이 되는 것이다. 아시다시피 리프 노드에서 루트 노드까지 검은 노드의 경유의 수가 달라지기 때문이다. 그렇지만 일단 25의 부모는 현재 존재하지 않는 상태이기 때문에 25의 왼쪽 자식의 큰 원을 기준으로 꺾어 내려가보도록 하겠다.



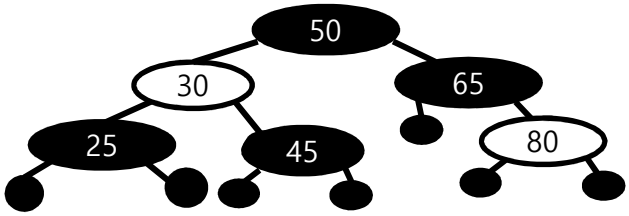
그럼 아래와 같이 25와 50의 색을 서로 바꿔주면 해결이 되는데 규칙을 지키는데 있어서 아직 배고프다. 01-2a의 규칙을 적용시켜서 다시 만들어두도록 하자.

문제발생!
01-2a 규칙



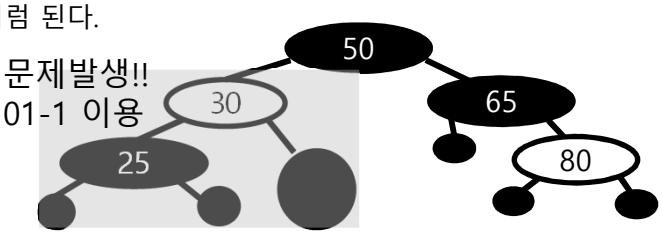


그럼 이제 25 노드, 25 노드의 색깔을 각각 검은색으로, 30 노드의 색깔을 빨간색으로 바꿔주도록 하겠다.

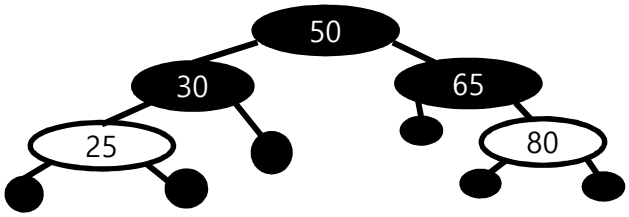


그러면 이제 레드블랙트리의 규칙을 지키게 되면서 종결을 짓게 되겠다.

4단계) 45를 삭제한다면...
여기서 왜 뜬금포 45를 삭제한다고 적어뒀을까? 바로 45를 없애고 나면 규칙이 벗어나게 된다는 점에 대해서 더더욱 설명을 해주고 싶기 때문이다. 일단 45를 없애게 된다면 아래처럼 된다.

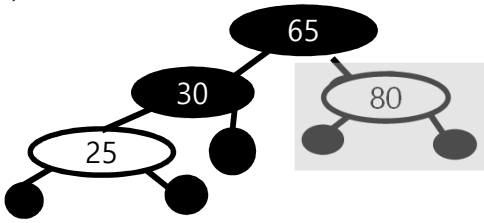


그러면 여기서 45가 삭제되었으니 45의 자식인 null 노드가 책임을 지게 된다. 그러면 25와 30에 대한 부분은 01-1에 대해 규칙이 어긋나게 된다. 그래서 25와 30의 색깔을 바꿔줌으로서 종결을 지으면 되겠다.



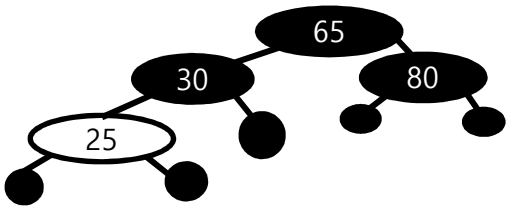
이처럼 삭제를 완료하고 나면 레드블랙트리 규칙에 대해서 지키게 되면서 종결이 난다.

5단계) 50을 삭제한다면...

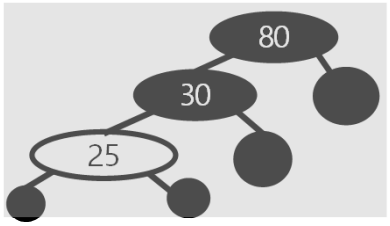


색상변경
작업 필요

블랙노드끼리 삭제를 하는 작업을 살펴보게 된다면 50은 자식이 현재 2명이 있기 때문에 우측 서브트리에서 최솟값인 65를 올리는 것이 현명하게 먹힌다. 그럼 여기서 65에 대해서 위로 올리는 대신해서 아래에 80이란 값을 색상 변경을 해주게 된다면 레드블랙트리의 원칙을 지키게 되면서 종결이 나게 된다.

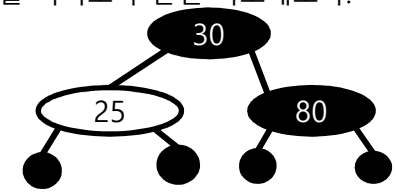


6단계) 65를 삭제한다면...

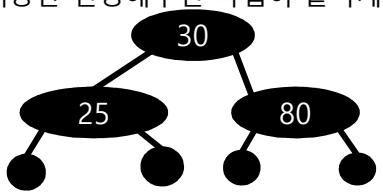


문제 발생!!
02-2a 이용

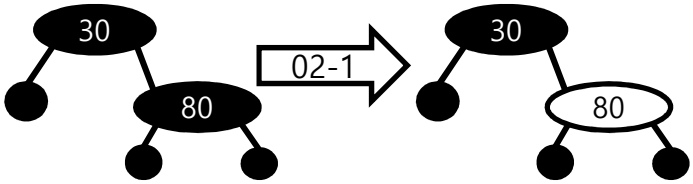
65를 지우게 된다면 65의 자식들은 2명이기 때문에 그 중에서 우측 노드인 80이 올라오게 된다. 그렇지만 여기서 문제가 발생을 하게 된다. 바로 02-2a번째 규칙에 따라서 조정을 해둘 필요가 있으니 이에 대해서 적용을 해서 레드블랙트리를 지키도록 한번 시도해보자.



분명 이렇게 변환이 되겠다. 그러면 최종 값인 25에 대해서 색상만 변경해주면 작업이 끝나게 된다.

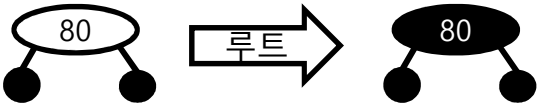


7단계) 25를 삭제한다면...



그러면 모두 검은색 트리가 나오는 02-1)과 같은 문제가 나오기 때문에 이에 형제 노드인 80에 대해서 색상을 입히면 되겠다.

8단계) 30이 삭제 된다면...



이제 80만 쓸쓸이 남게 되어서 이도 알아서 검은색으로 되면서 루트 노드의 역할을 충실하게 되면서 종결나게 된다. 그리고 80이란 노드가 없어지면 결국에 모든 노드들이 없어져서 이 부분에 대해서는 새로운 값을 추가함으로서 재조정을 하는 방법이 있다.

이상 레드블랙트리의 삭제 알고리즘에 대해서 자세히 살펴봤다. 생각보다 많이 어렵게 느껴질지 모를지 언정 시간이 나게 된다면 언제든지 연습을 해보면서 공부를 할 기회를 가졌으면 좋겠다. 다음으로는 B-트리에 대해서 살펴볼 것이다. B-트리는 레드블랙트리보다 할만해서 금방 정리된다.